

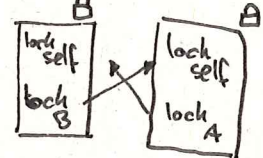
Mem. reordering: processors allowed to change order of single operations if they (apparently) do not modify program (do not affect semantics of sequential code)

→ eg. due to pipeline/cache
no guarantee of global ordering

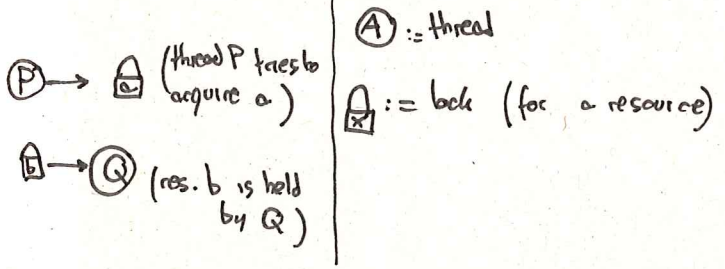
△ while(*!=*);
↳ while(true); //this causes problems, OP loop

Dead Lock

eg. Bank transfer



two or more processes are mutually blocked because each process waits for another of these processes to proceed.



DL occurs when the graph of dependences between threads-locks contains a CYCLE

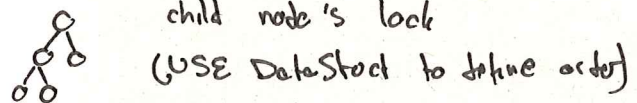
- DL detection simply looks for that
- DL avoidance can be achieved by
 - two phase locking
 - resource ordering

RES. ORDERING: start acquiring resources with bigger # order (or thread w/ bigger ID) and set free the smaller

→ if # ordering not available, generate it

You can document order for thread using, eg. "when doing this, never try to acquire this DL while holding that DL"

for trees: if holding a node's lock, you may only acquire child node's lock



(USE DataStruct to define order)

Interleaving: program is not executed in sequential order like it was programmed (thread splitted execution)

synchronized: keyword in java used to avoid data races & inconsistency

volatile (used to declare variable, but ~~not~~ for experts, avoid)

atomic operations

when doing something that depends on a Read (eg. WAR/RAW) make sure they cannot be split.

STARVATION:



repeated but unsuccessful attempt of a recently unblocked process to continue its execution.

READERS / WRITERS LOCK

- RAR → no problem (can allow multiple)
- WAR/RAW → X problem
- WAW → X problem

if lockup doesn't modify DataS, can allow more to happen simultaneously

use R/W lock:

states

- not-held
- held - writing
- held - reading

operations

- new() // not locked
- acquire_write() *1
- release_write()
- acquire_read() *2
- release_read()

*1 only possible if not held.
if held by readers, don't allow new readers to acquire (else starvation)

*2 always possible if held by readers.
increase readers count by 1

use java.util.concurrent.locks.ReentrantReadWriteLock

Semaphores



normal locks good for mutex and atomicity

↳ allow threads to communicate about changes

has INT, is DataS, initial value $S \geq 0$ (means how many can happen simultaneously)

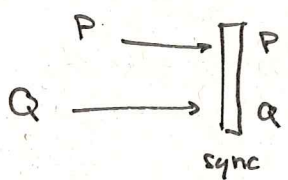
- acquire(S) $S--$
- release(S) $S++$

$S = 0 \rightarrow$ locked

$S = 1 \rightarrow$ unlocked

$S > 0 \rightarrow S$ threads can be let into critical section

SEMA can be used for Rendezvous



P(wait/acquire)
V(signal/release)

Semaphores P-arrived, Q-arrived

	P	Q
init	P-arrived = 0	Q-arrived = 0
pre	...	...
sync	P-arrived.release Q-arrived.acquire	Q-arrived.release P-arrived.acquire
post	...	...

releasing before acquiring avoids DL
✓ Best solution

Barrier like Rendezvous but for multiple processes

after a few trials gone wrong: SOLUTION

2-PHASE BARRIER

```

init {
    mutex = 1, barrier1 = 0, barrier2 = 1, count = 0
}

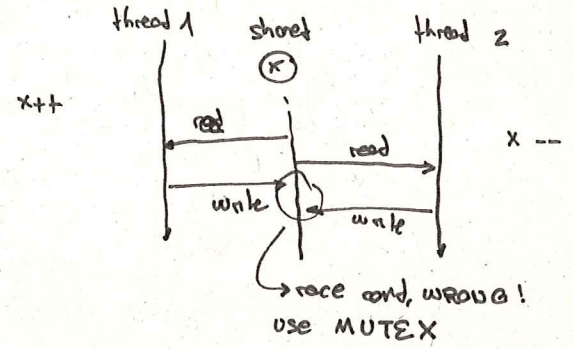
barrier {
    acquire(mutex)
    count++;
    if (count == n)
        acquire(barrier2); release(barrier1);
    release(mutex);

    acquire(barrier1); release(barrier1); // b1=1, b2=0 for all processes

    acquire(mutex)
    count--;
    if (count == 0)
        acquire(barrier1); release(barrier2);
    signal(mutex);

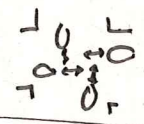
    acquire(barrier2); release(barrier2) // b2=1, b1=0 for all processes
}
    
```

Race condition: execution order of code, or time propagation for these variable impacts it's value → WE DON'T WANT IT!

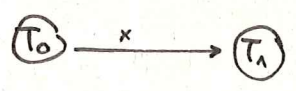


L15 - SYNC beyond locks 2

Live lock: competing threads are able to detect potential DL but make no progress while trying to solve it



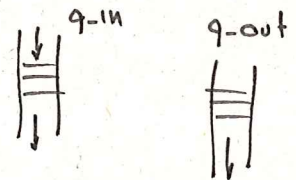
PRODUCER / CONSUMER PATTERN



T₀ computes x, gives it to T₁ that uses it

```

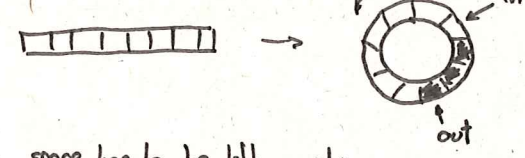
input = q-in.Dequeue();
output = do-Something(input);
q-out.Enqueue(output);
    
```



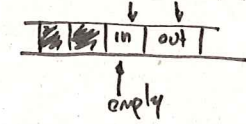
semaphores are not sufficient for P/C queues

- unstructured (require discipline, easy to DL)

Circular buffer



one space has to be left empty



try { -- }

finally { -- // executes also if try fails and quits with exception }

Thread.sleep(time); → for attender if thread: maybe do while(true) the spreca CLK

MONITOR: can temporarily escape from lock when waiting on a condition

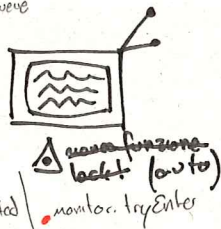
is abstract DataStr equipped with op. that run in mutex they provide, other than mutex:

- check for condition: if doesn't hold → release monitor lock
- wait for it to be True
- signals, avoids loops (busy loops, CLK)

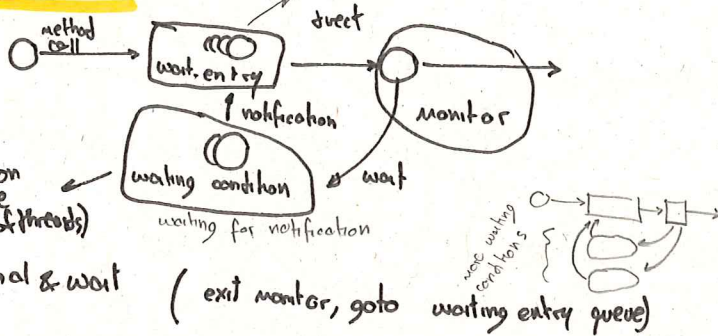
PAR-CONT

- wait() // until signaled
- notify() // wakes up 1 thread
- notifyAll()

release mutex
move to wait queue
stop thread



Monitor Queues



- signal & wait (exit monitor, goto waiting entry queue)
- signal & continue (continues processing)

in JAVA:

final Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();

- await()
 - signal()
 - signalAll()
- ex pulse

sleepy barber variant: used additional counters to know if processes are waiting, so that it can avoid sending signals if not useful

- Critical section:
- mutex (must not be interleaved)
 - freedom from DL starvation

IRQ.off → interrupt request
critical section
IRQ.on

```
acquire(monitorLock);
while (!conditionMet) {
    wait() // on this monitor,
           // under condition
           // variable
}
// crit. section
signal(cv); // condition variable
release(monitorLock);
```

L16 - Behind Locks Implementation

CRIT. SECTIONS

events & precedence

thread P produces events $p_0, p_1, \dots$

j^{th} occurrence of event i
(eg 3rd iteration loop, event 5)
 $p_i^j = p_s^3$

precedence (total order relation)

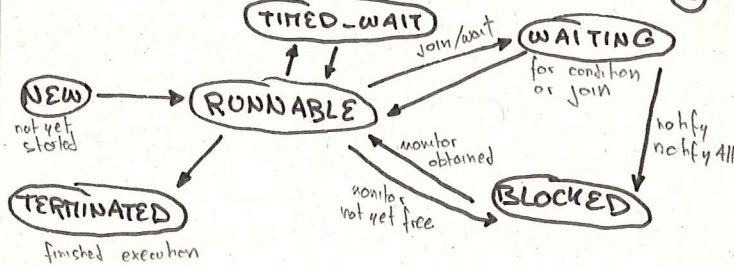
$a \rightarrow b$ (a occurs before b)

Interval: $(a_0, a_1) \Rightarrow a_0 \rightarrow a_1$

$I_A = (a_0, a_1), I_B = (b_0, b_1), I_A \rightarrow I_B \Rightarrow a_1 \rightarrow b_0$

" I_A precedes I_B ", else "they are concurrent"

Thread STATES



Conditions, always associated with Lock

P/C queue with monitors lock

```
Class Queue {
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    Queue(int s) { // constructor
        size = s;
        buf = new long[size];
    }
}
```

```
void Enqueue(long x) {
    lock.lock();
    while (isFull())
        try { notFull.await(); }
    catch (InterruptedException) {}
    do Enqueue(x);
    notEmpty.signal();
    lock.unlock();
}

long Dequeue() {
    long x;
    lock.lock();
    while (isEmpty())
        try { notEmpty.await(); }
    catch (InterruptedException) {}
    x = doDequeue();
    notFull.signal();
    lock.unlock();
    return x;
}
```

Decker's Alg for crited sections

bool wantP=false, wantQ=false, int turn=1

PROCESS P	PROCESS Q
loop	
non crit section	
wantP = true	wantQ
while (wantQ)	while (wantP)
wantP = false	wantQ
await (turn==1)	awaitP (turn==2)
wantP = true	wantQ
critical section	
turn = 2	turn = 1
wantP = false	wantQ

more concise: Peterson Lock

P=1, Q=2, bool[] flag = false[], int victim=1

```
Process P
loop
    non-crit-section
    flag[P] = true
    victim = P
    await (!flag[Q] || victim != P) → no interest?
    crit-section
    flag[P] = false
    → I GO
```

filter Lock: extension of Peterson's lock to multi processes

```
int [# threads] level, int [# threads] victim
lock: for (int i=1; i<n; i++)
    level[i] = i; victim[i] = i;
    while (exists me: level[me] > i && victim[i] == me) {}
    unlock;
    level[me] = 0;
```

Hardware support: Read-Modify-Write OP

- TAS (test and SET)
- CAS (compare and SWAP)
- load linked / store conditional

→ enable mutex implementation with O(1)
→ lock free programming

Atomic TAS(memref s)
 $\begin{cases} \text{if (mem[s] == 0)} \\ \text{mem[s] = 1; return true;} \\ \text{else return false;} \end{cases}$

Atomic CAS(memref s, int old, int new)
 $\begin{cases} \text{oldval = mem[s];} \\ \text{if (oldval == old)} \\ \text{mem[s] = new;} \\ \text{return oldval;} \end{cases}$

implementation of spinlock:

init(lock)
lock = 0;
Acquire(lock)
while !TAS(lock); // wait
Release(lock)
lock = 0;

init
→ "
Acquire(lock)
repeat
res = CAS(lock, 0, 1);
until res == 0;
Release(lock)
CAS(lock, 1, 0)

L17 - Without LOCKS

putting to sleep has also "negative" side: higher wake up latency. possible solution: Hybrid: spinlock at beginning, change in schedule after some time

LOCK - FREE

- lock freedom: at least 1 algorithm makes progress if others are running
- wait freedom: all obj. eventually make progress (no starv)

	non-blocking	blocking
all make progress	wait-free per-thread progress	starv-free
one makes progress	lock-free syst-wide progress	DL-free

→ block/suspension of thread doesn't affect others (with also blocking or suspension)

use CAS: read - check if no change (you are exclusive) → write

the filter lock is not fair: it's not (FIFO)

→ bakery (Post) Algorithm: idea - take a number ticket

2 processes: int np = 0, ng = 0

Process P: loop
non_crit_section
np = ng + 1
await (ng == 0 || np < ng)
crit_section
np = 0

q doesn't require access or I got earlier ticket

SPINLOCK: normal lock that uses bop (busy waiting) to check if lock has been set free. usually inefficient, one can use specific APIs. → don't have notification system

Atomic in java: java.util.concurrent.atomic.AtomicBoolean

AtomicBoolean state = new AtomicBoolean(false);

lock {
• set
• get
• compareAndSet (boolean expect, bool update)
• getAndSet (bool newvalue)

Backoff: too many threads trying to access same resource → slows down every thing.

solution: put them to sleep with random duration (increase duration each time resource is not free)

* see p2

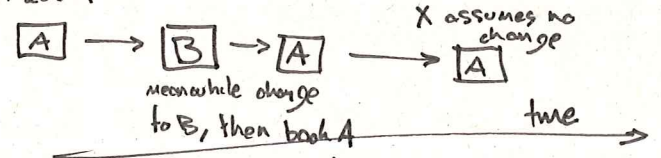
called Locks with scheduling (puts same to sleep.)

→ semaphores, ~~and~~ monitors use it

Locks - pessimistic by design performance: overhead, contended case = slowdown, bottleneck if delay in CS, everyone suffers

ABA problem: activity doesn't recognize that mem location was temporarily ~~been~~ modified by another activity and therefore ~~unwittingly~~ wrongly assumes it hasn't been changed.

X sees A



- solutions
- DCAS (double CAS)
 - pointer tagging (save last bits of address, now 1/32 chance of fail)

• hazard pointer (before reading, makes it as dangerous)

PAR.P CONT

SCHEDULER - heart of OS

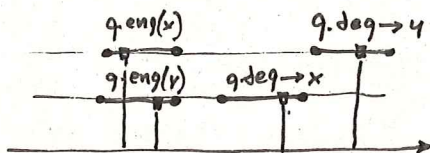
L18 - Concurrency theory

LINEARIZABILITY

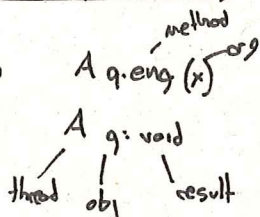
- each method should appear to take effect instantaneously between invocations and response events

→ obj for which this is true for all possible executions is called linearizable

(simply see if it's possible that with a certain order things happen that way)



Method calls → 2 events: invocation response



History: sequence of invoc & resp.

$$H = \begin{cases} A \text{ q.eng}(x) \\ A \text{ q: void} \\ A \text{ q.eng}(x) \end{cases}$$
 matches, if invoc & resp match
 → pending (no resp yet)

sequential if no method calls interleave

History is complete if no pending

Projections: Obj-proj Thread-proj

$H|_q = (\text{all calls \& resp of } q)$
 $H|_A = (\text{ " " of } A)$

History:

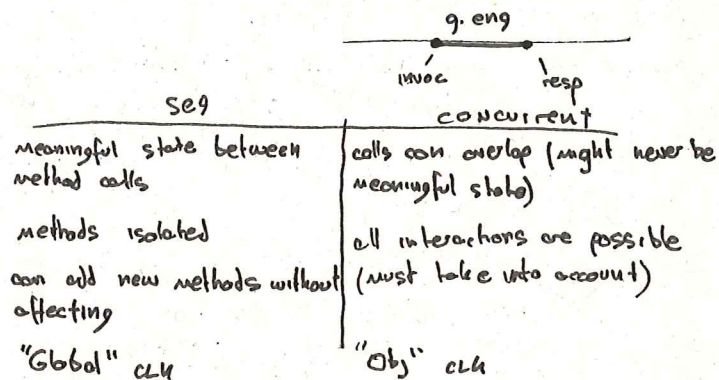
- well formed (per thread projection is sequential)
- equivalent (each u is equivalent)
- legal (pre/post cond tell what is legal for obj)
- if every obj is, then also History

VOLATILE: ask java to keep variable up to date

Series matters, volatile is volatile, so volatile, volatile is volatile

Obj → container for data with set of methods to modify state

Method call → start with invocation ends with response



Precedence: A precedes B iff A.response prec. B.invoc (else overlap)

History Linearizable ⇔ can be extended to history G

- appending $x \geq 0$ responses to pending invocations (that happened)
- discarding $y \geq 0$ pending invoc that didn't take effect

→ $G \subseteq S$ (S respects order of G, and adds a few constraints)

Composability: H linearizable ⇔ $\forall x (x \text{ obj}, H|x \text{ linearizable})$

SEQ. CONSISTENCY

H is sequentially cons ⇔ same, but the ... ?

"such that" is not required

CONSENSUS

all thread decide same value
 (common decision is one of the inputs)

public interface Consensus<T> {
T decide (T value);
}

RTW (read-modify-write)

```
getAndSomeOp() {
  int prior = this.value;
  this.value = someOP(this.value);
  return prior;
}
```

L19 - From Java to C

C is more in touch with basic HW
• memory allocation (& management)

Java hides many details for OS woking

OPENCL, OPENMP, MPI

possible errors while compiling:

- preprocessor (miss. include files)
- parser (syntax errors)
- linker (miss. libraries)

C preprocessor (cpp)

↳ takes # statements and substitutes in the code
(#define res 255) → don't define functions

#ifdef name
code segment 1

#else
code seg 2

#endif

typedef short int smallNumber

↳ (gives names to types)

typedef char boolean;

#define FALSE 0

#define TRUE 1

POINTER * : direct mem address

& → provides pointer to data obj (address operator)

* → provides data obj to a pointer (dereference op)

int i = 5;

int* ptr = &i;

int j = *ptr; // j = 5

void pointer: can be assigned to every type

L20 - OPEN MP

concurrency: multiple tasks are logically active at once



parallelism: multiple tasks are actually active at one time



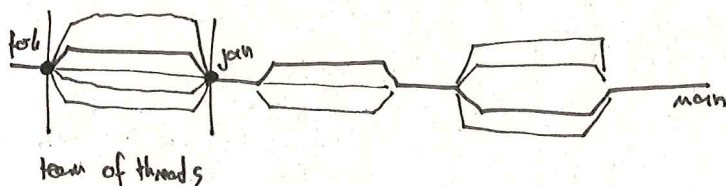
Syntax

compiler directives

```
#pragma omp construct { clauses }
# parallel num_threads (4)
```

threads communicate via shared variables

fork-join



false sharing (cache inconsistency when saving)

JAVA

OOP
polymorphism
autom. memory management
references
collection of classes

C

sequential
united
• malloc, free
• pointer (mem. address)
collection of functions

every data obj has

type
(name)
address
value
size
visibility
lifetime

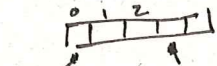
→ = .

Array: int vec[3];

doesn't remember how large they are

size of = # bytes. do $\frac{\text{sizeof } A}{\text{sizeof } A[0]}$

no checks for bounds



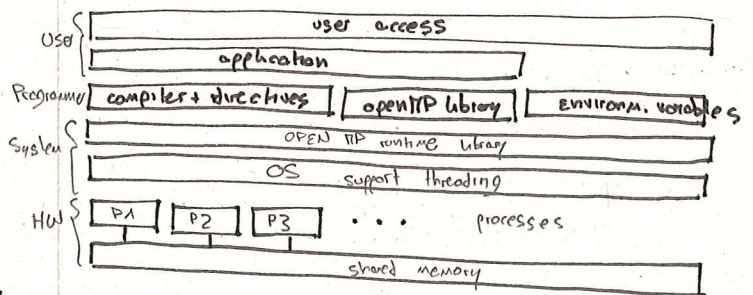
(s = size of 1 cell)

↳ they are pointers!

int* a = (int*) malloc (len * sizeof(int));

struct (like Java, C#)

OPEN MP - api for multi-thread applcat.



LIST

```
omp_set_num_threads(8); // how many
#pragma omp parallel { // what's inside will be
    // executed parallel
    int id = omp_get_thread_num(); // inside block, gets
    // unique ID
}
```

```
int omp_get_num_threads(); // # in current team
double omp_get_wtime(); // wall CLK in seconds
```

```
#pragma omp barrier // simple barrier
```

```
#pragma omp critical (res += consume B;) // can only be executed by 1 at
// time
```

```
#pragma omp atomic //
```

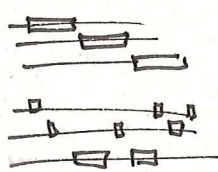
```
#pragma omp for
for (i = 0; ...)
```

PAR.P CONT

static vs dynamic

fixed size chunks
variable

schedule



storage attributes

- shared
- private
- first private (in)
- last private (out)
- default

private var are not initialized, and can't be used outside scope

means local variable (else variable is shared by all)

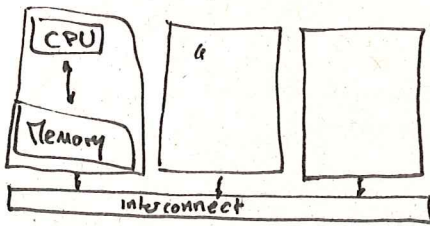
propagated in/out

task: independent unit of work

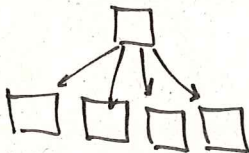
- code to execute
- data environment
- control variables

LZA - OPEN MP & MPI

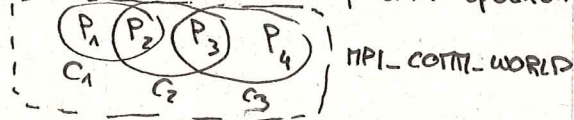
distributed memory system



SPTD (single program, multiple data)



MPI communicators: defines comm. domain of comm. operation



since many diff messages can be sent between 2 guys, tag is used to differentiate.

receiver can get msg without knowing (sender, tag, amount data)

Reception status

SEND
Synchronous mess. passing → waits until complete mess can be accepted by receiving process (before sending)

RECEIVE waits until gets message
used for data transfer + sync

Asynchronous mess. passing

send/receive do not wait

requires local storage for messages.

about communication
S - R

Reduction

#pragma omp parallel shared (A, B, C) private (id) {

master // code only executed by master thread (id == 0)
single // everyone do 1 single thread section

#pragma omp lock

thread private (variables) // provides private copy of global variable for each thread

#pragma omp parallel for reduction (+:sum)

local copy is made depending on "op"
+/- → 0
* → 1
min → largest
max → smallest

cs. storage

Locks

omp_init_lock();
omp_set_lock();
omp_unset_lock();
omp_destroy_lock();

message passing: mechanism to send/receive messages

MPI (mess. passing interface)

use header <mpi.h>

MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &size);
rank

id, number

points to data to be sent

bytes to be sent

type

int MPI_Send (void *msg_buf, int msg_size, MPI_Datatype msg_type, int dest, int tag, MPI_Comm communicator);
destination data identification tag

have to watch!

int MPI_Recv (void *msg_buf, int buf_size, MPI_Datatype msg_type, int source, int tag, MPI_Comm communicator, MPI_Status *status_p);
// buf size
// source process id
// status/error

eg. MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

Blocking/non ~

return after local actions are complete (only then) → return immediately

about local handling
Sending/receiving

MPI_Send

- blocking
- sync/async depends on implementation

MPI_Recv

Parallelization steps

- 1) partition solution into tasks
- 2) identify comm. channels between tasks
- 3) aggregate tasks into composite
- 4) map it to cores

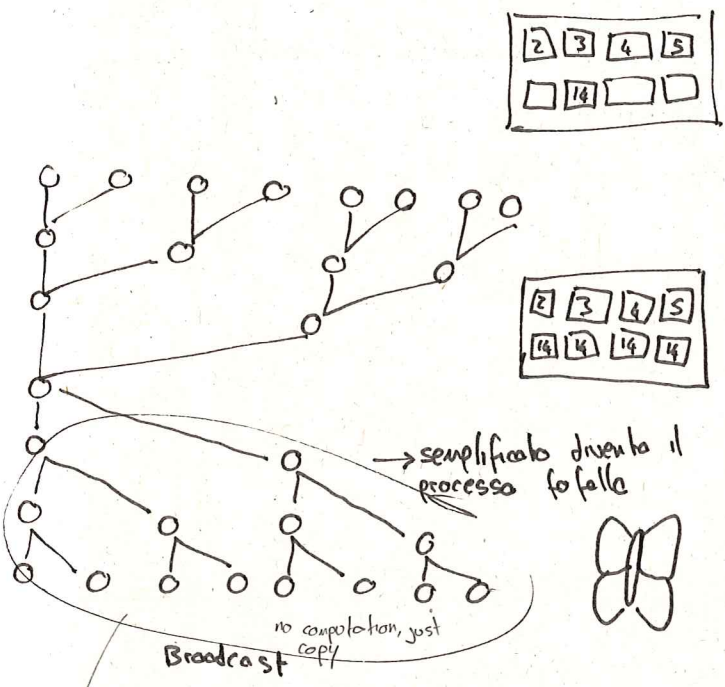
eg. can be used for Σ .

COLLECTIVE COMMUNICATION

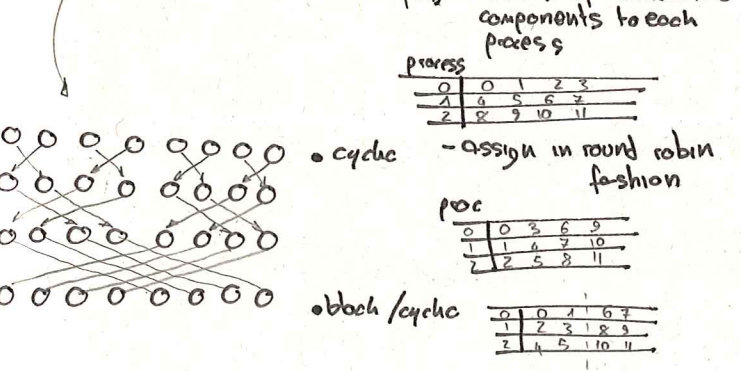
eg. Group comm

- up to here $\rightarrow$ point2point comm
- you can also send mess between groups of processors

eg. MPI_Bcast
Reduce
Allreduce
Scatter
Gather
...



Partitioning scheme

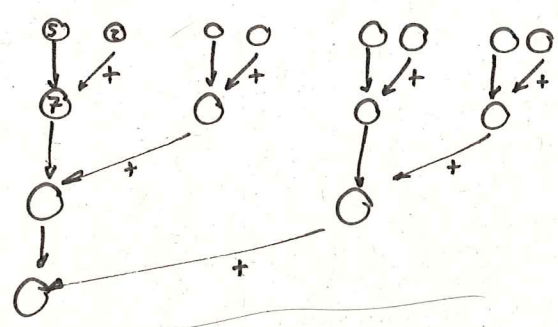


there also are variations for block/non ~ sync/as ~ buffered/non ~

MPI_Bsend
Ssend
Rsend ...

most implementation only allow process 0 to access important stuff (eg stdin) / read data

A tree structured sum



MPI_Reduce

```

void* input_data_p,
void* output_data_p,
int count,
MPI_Datatype datatype,
MPI_Op operator,
MPI_Comm comm);
  
```

MPI_MAX/MIN
- SUM/PROD
...

eg. MPI_Reduce(&local_int, &global_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)

all processes in communicator must call same funct
args passed must be compatible

MPI_Allreduce (copies results in all) ✓

Broadcast (data belonging to one of the processes is sent to all processes in communicator)

Scatter (reads in an entire vector, then sends only the required parts to the other processes)

Gather (collect all components of vector into destination process, then this can process all components)

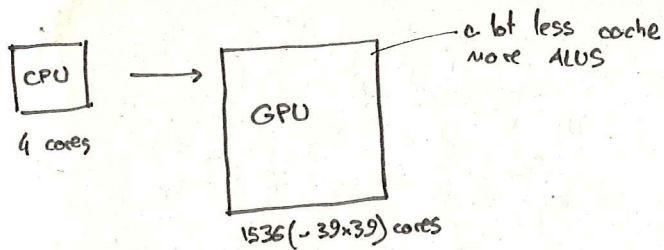
Allgather = gather + broadcast (save everything in everyone)

MPI_Gather

```

void* send_buf_p,
int send_count,
MPI_Datatype send_type,
void* recv_buf_p,
int recv_count,
MPI_Datatype recv_type,
int dest_proc,
MPI_Comm comm
  
```

You can use this to create a system for matrix mult.



500 vs 4500 (9x) gflops (Giga float op/s)
40 vs 170 GB/s bandwidth

→ GPU power makes them interesting for general purpose computation

in the past was very difficult to use them

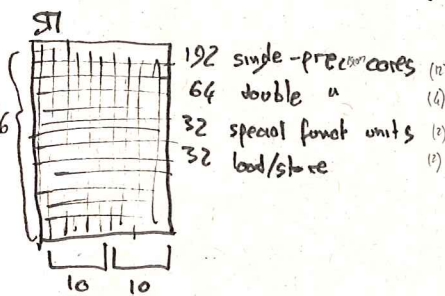
- rapidly evolving
- secret
- inherent parallel

CPU: focus on low latency
(out of order and speculative execution)

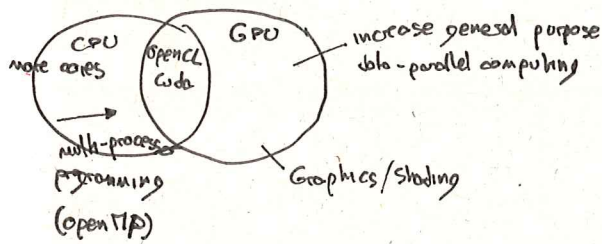
GPU: focus on throughput

terminology: SPA(device) streaming processor array
SP
SP

streaming multiprocessor



Heterogeneous computing



CUDA (compute unified Device architecture)
• general purpose programming model

OPEN CL → open computing language
(open standard for heterogeneous programming)

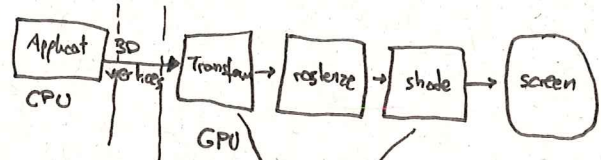
- cross platform parallel computing api, C-like language
↳ large fs under range than CUDA
(less peak performance than specific CUDA)
- language specification for kernels
(executed by compute device)

• we can no longer increase CLK frequency

- E consumption
- heat
- too long path

more cores on chip requires cores smaller, simpler

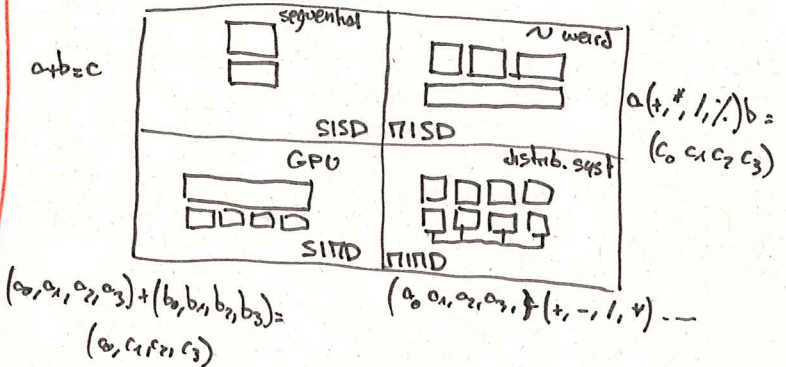
GPU: ^{es.} fixed graphics pipeline



can do in parallel with vectors!

Granularity: coarse (process computes on a large block of data before communicating)
fine (individual process quite small, more communication)

Flynn's Taxonomy



vector processing
SIMT (multiple threads)
VLIW (very long instr. word)

objective for today

L23 - GPGPU

Host program written in High-level language (C/C++)
→ kernels written in OPENCL

Vocabulary:

- platform (vendor's implementation of OPENCL)
↳ manages collection of devices
- device: piece of HW accessible via vendor's platform
- context: collection of devices (same platform) aimed at working together + command queue (with work to be done)
- kernel: smallest unit of execution for device // work item (C funct. + part of program)
- program: collection of kernels (stored on file)
- work item: instance of kernel at runtime. (runs in work group), has unique ID
- work group: collection of work items (unique work ID)

DATA PARALLELISM: define N-D computation domain
↳ it defines tot # of work items

Δ problems that map well (independent, sorting (radix, bitonic), lin. algebra) vs not so well (sequential, dependent)

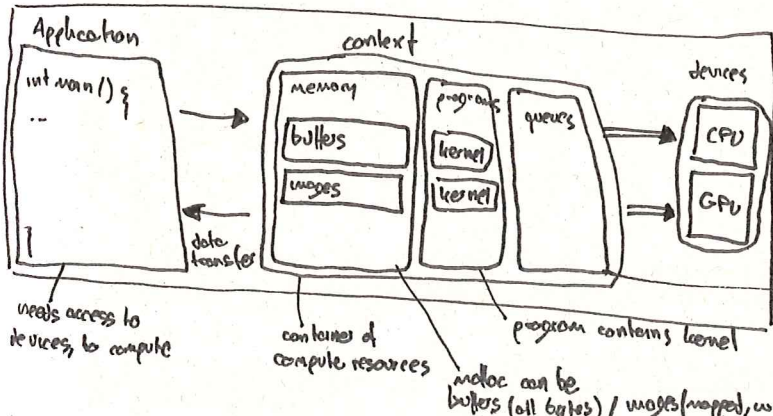
Array of work items (execute kernels)
work items within work group cooperate (shared mem/atomic operations) **SIMD**

Domain config: 1D (eg. elem. in array)
2D (eg. image)
3D (eg. voxels)

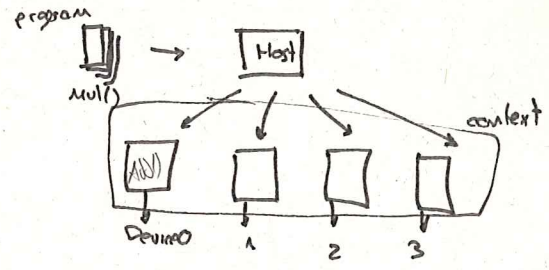
Host: • create & manage device context - associate work queues
• memory alloc
• kernel launch
↳ compiled at runtime (by host)

Data types: scalar (normal)
vector (es. bool, n)

there are object creation management methods
extensions (-kernel, void ...)



OPEN CL & CARDS



Structure of OPENCL program

- 1) get info about platform and devices available
- 2) collect devices in context
- 3) create opencl commandqueue in context
- 4) " mem. buffer on device
- 5) transfer data from host to device mem. buff
- 6) create kernel
- 7) build/compile kernel
- 8) create OPENCL kernel obj
- 9) set kernel args
- 10) execute kernel
- 11) read kernel mem and copy to host mem

`int id = get_global_id(0);` // looks for ID per square case
dere calculate (index in vector array)
CPU coordina, GPU lavora/calcola

API

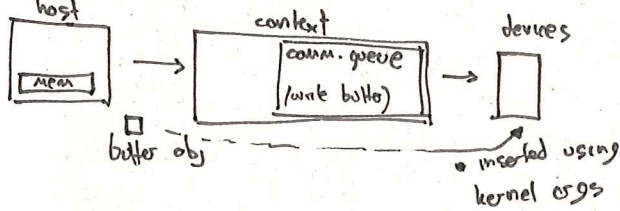
- //get platform IDs ①
`cl_platform_id * platforms = (cl_platform_id *) malloc(sizeof(cl_platform_id) * length);`
`res = clGetPlatformIDs(len, platforms, &nPlatforms);`
- //get platform info
`res = clGetPlatformInfo(platforms[i], CL_PLATFORM_EXTENSIONS, 0, null, &ext.size);`
- //get device ID, info
`cl_device_id * devIds = (cl_device_id *) malloc(sizeof(cl_device_id) * length);`
`res = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_GPU, length, devIds, &nDevices);`
- //create context ②
`cl_context context = clCreateContext(...)`
- //create command queue ③
`cl_command_queue c = clCreateCommandQueue(context, devIds[0], 0, &err);`
- //create buffers ④ ⑤
`cl_mem * buffers = (cl_mem *) malloc(sizeof(cl_mem) * nBuffers);`
`res = clCreateBuffer(context, CL_MEM_READ_WRITE, size, 0, &err);`
- //create & compile program ⑥ ⑦
`cl_program program = clCreateProgramWithSource(context, nSources, sourceStrings, &err);`
`res = clBuildProgram(program, 0, 0, 0, 0, &err);`
- //determine kernel to execute ⑧
`cl_kernel kernel = clCreateKernel(program, kernelName, &err);`
- //set kernel args ⑨
- //schedule kernel for execution ⑩
`res = clEnqueueTask(kernel, c, 0, 0, 0, &err);`
- //read result ⑪

* SEE APIS

code can be compiled for different devices (CPU, GPU, ...)

mem. transfer: use kernel args

memory transfer



pointer args can be everything but private
global variables must be const

EVENTS

- host notification (eg. command completion)
- command sync
- profiling

can be associated with callback fct (called when event happens)

usually, the elements in context command queue are sent to devices in out of order. You can't control it.
but you can constraint a command to wait for another to be executed

you can obtain profiling info

host mem can be mapped to device mem, to save all data transfer
(very efficient, but not always best choice)

KERNEL LVL SYNC MECHANISM

- memory model uses relaxed consistency
- if one data is visible to one work item, it's not guaranteed to be seen by all in work group
 - you need sync for that

L25 - OpenCL Case studies

Map & reduce

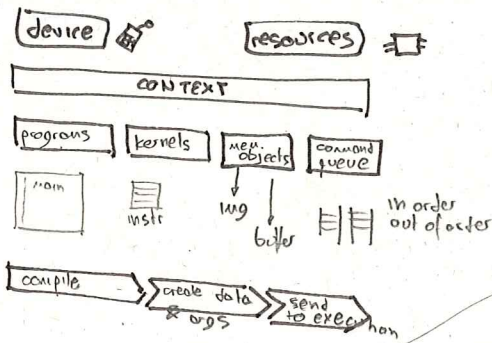
- strategy for implementing parallel alg
- map: A master worker takes problem input, divides into subproblems, and distributes to threads
 - reduce: master worker collects solution of workers and combines them to get out final solution

D&C

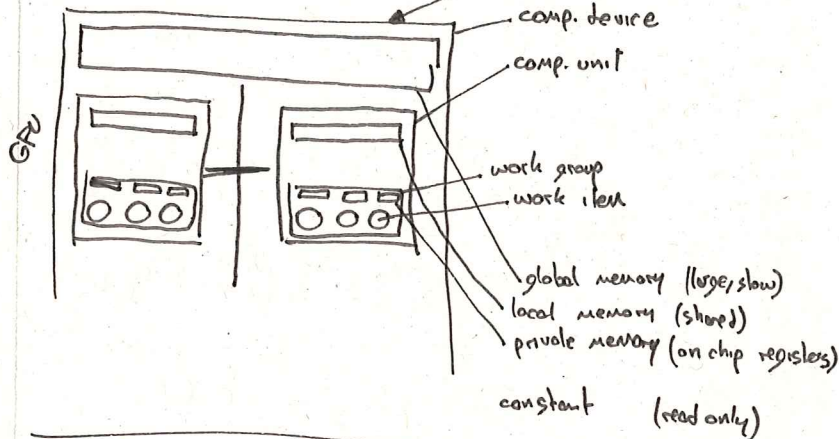
OpenCL doesn't feature dynamic malloc! You cannot know in advance # of value-pairs

- 1) each work item in work group performs mapping
- 2) barrier prevents further execution until done
- 3) local reduction: each local ID == 0 work item (first in group) performs reduction
- 4) global barrier
- 5) work item with global ID == 0 performs final reduction

OPENCL program flow



OPENCL memory model

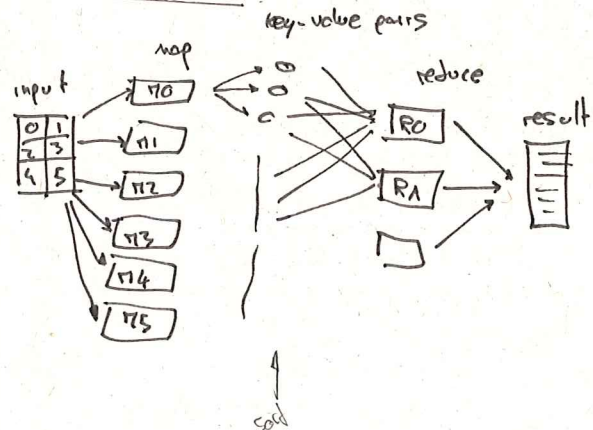


Global & local: #global work items limited by size_t
#work items per group limited by (max work group size)
↳ have 2 IDs: global & local

work group size: clGetKernelWorkGroupInfo

Memory organisation: loc. mem organized into mem banks (usually 16/32)

literals



Sorting

Heaps & Verges $\rightarrow O(n \log n)$ worst

Quicks

$\rightarrow$ average $O(n \log n)$, $O(n^2)$ worst

impossible to go lower!

permutations: $n!$

these are all comparison sorting.

At best, we eliminate $\frac{1}{2}$ of possibilities while searching for something

binary: each question $\rightarrow 2$ outcomes

no duplicates / redundant questions

every answer is leaf $\rightarrow$ tree has to be big enough to contain all

$\Rightarrow$ no alg can be better than this running time!

comparison

height of bin tree with $n!$ leaves

$\hookrightarrow$ height $\Omega(n \log n)$

lower bound on height:

bin tree with L leaves has at least height $\log_2 L$

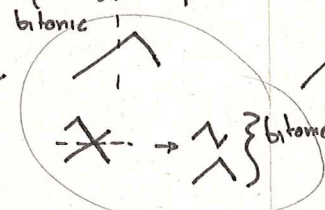
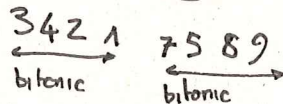
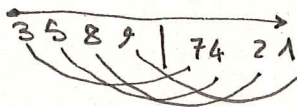
$$n \geq \log_2(n!) = \log_2(n(n-1)(n-2)\dots) = \log_2 n + \log_2(n-1) + \dots + \log_2 1$$

$$\geq \log_2 n + \log_2(n-1) + \dots + \log_2\left(\frac{n}{2}\right)$$

$$\geq \left(\frac{n}{2}\right) \log_2(n/2) = \frac{n}{2} (\log_2 n - \log_2 2) = \frac{n}{2} (\log_2 n - 1)$$

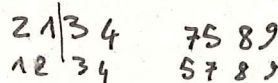
$$\rightarrow \Omega(n \log n)$$

Binary split: divide bitonic in 2 - compare & swap



since it was bitonic, every element in first partition is smaller than element in second

recursively "bitonic splitting" will sort



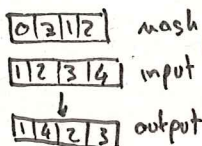
forming bitonic from scratch:

1) pairs of adjacent # already bitonic $\rightarrow$ merge in larger sequence

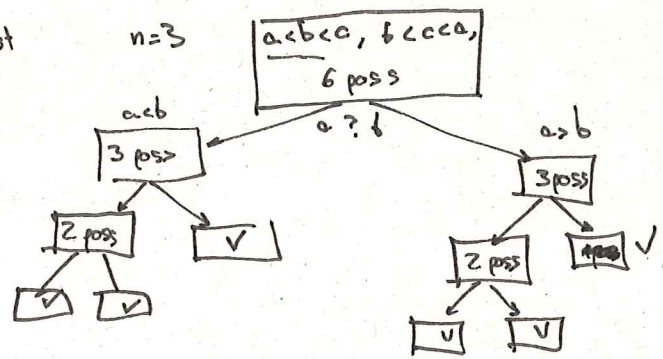
2) compare first 2 elem. if $1^{st} > 2^{nd}$, swap. ensure ascending order

3) compare second 2 elem. if $2^{nd} > 1^{st}$, swap. ensure descending.

shuffle: take elements in order from mesh



$\rightarrow$ node tree: each node remaining poss



Horrible alg $\Omega(n^2) \rightarrow$ bogo, stooge

simple $O(n^2) \rightarrow$ insertion, selection, bubble

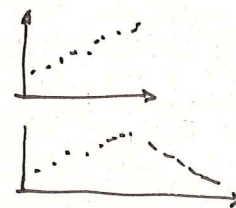
fancier $O(n \log n) \rightarrow$ heap, quick, merge

specialized: $O(n) \rightarrow$ radix, bucket

Bitonic: $O(n \log^2 n)$ - sequential

$O(\log^2 n)$ - parallel (worst=average=best)

bitonic sequence



monotone $\uparrow \downarrow$

sign of gradient changes 1 at most (max 1 local max & 1 local min)

wraps around (always 1 be max & min)

Good vocabulary

wiki on "monotonic sync"

threads interleaved inconsistent state

volatile: becomes visible to readers once writer has finished (otherwise they can see some non updated value)

massimo: 4 pagine per pagina ($\frac{1}{4}$)

	control	hardw	power	freedom
1e core	complex (bigger)		inefficient	flexibility/perform
1h core	easy		✓	restrictive - difficult reprogramming

PARALLEL

more resources to solve faster

(reference to exact platform)

work
↓ ↓ ↓
resources
perform/orchestration

CONCURRENT

execution of activities at same time, coordination

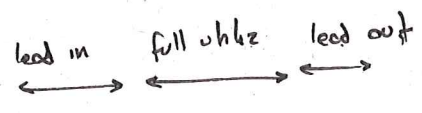


correctly/efficiently manage resources

DISTRIBUTED

physical separation

Architettura von Neuman (data & program in same memory)



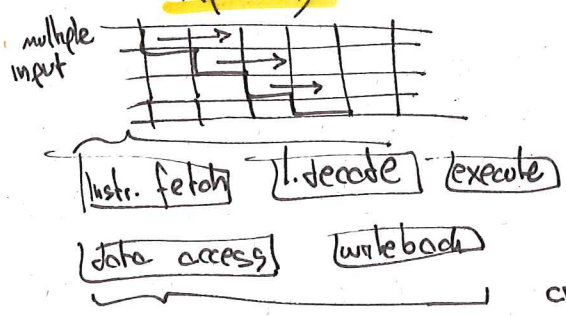
1 item/TU
TU = slower stage

Parallel strategies:

Vectorization

(do n at a time)
load → ⊕ → store

Pipelining



Sele. stages
CPO funct. units

ILP (inst. level parallel)

se le cose non dipendono tra loro (istruz. livello più basso)
fare in parallelo

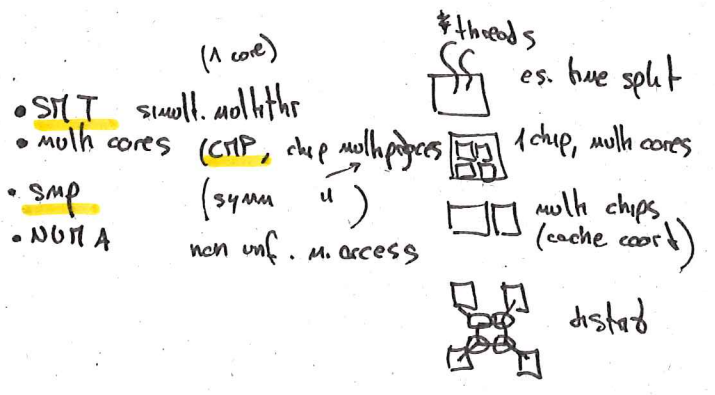
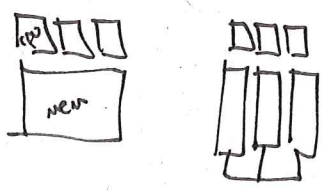
ADD \$3W5C
LD \$10WP NZK

Moore: double every 18 months (#transistor)

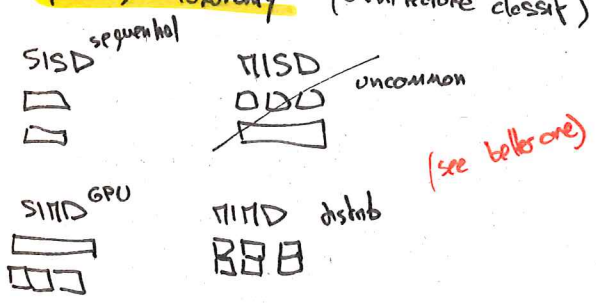
Δ power dissipation (worn) → add cores
Δ memory access speed

ARCHITECTURES

shared/distr. memory



Flynn's Taxonomy (architecture classif)



Formulas

perform (time) $T_p \approx T_1/p$

speedup $S_p \leq \frac{T_1}{T_p}$ (if linear, perfect, no overheads) extra computation

Amotahl (b. sequential)

$T_p = T_1(b + \frac{1-b}{p})$

$S_p \leq \frac{p}{1 + b(p-1)}$

$S_p \leq \frac{1}{b + \frac{1-b}{p}}$

serial part = problem (overheads)

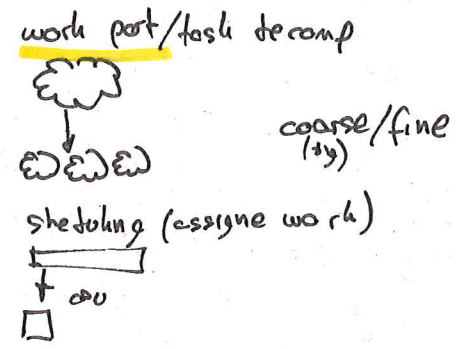
Gustafson (run time const)

$$\begin{cases} T_1 = p(1-b)T_p + bT_p \\ S_p = p - b(p-1) \end{cases}$$

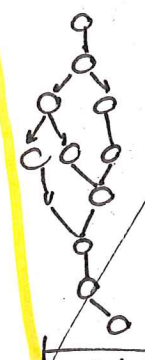
$T_{seq} = \text{seq work}$

TASK PAR:
DATA PAR:

user expresses tasks (genetic, not product)
" operation, systems splits



Task: spawn tasks graph



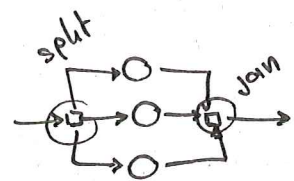
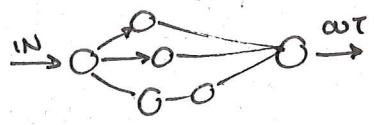
dynamic! but depends on previous

sequential (span, critical path)

parallelizable "wider is better"

DAG (directed acyclic graph)

if static: data flow



scheduling (greedy)

with $T_p \leq (T_1/p) + (T_{seq})$

w_{par} about w_{seq}

$\frac{T_1 - T_{seq}}{p} + T_{seq}$
better

D&C: recursive splitting

MANAGING STATE

- mutability (data doesn't change)
- iso state
- mutable (potential errors $\triangle$) (protect state to avoid concurrent access)

RPT (version poss. method)

$$a.b = 2a. \frac{1}{2}b$$

$$= a + (2a. \frac{b-1}{2})$$

to avoid failure, ensure correctness of code

USE Invariants

* 15	x 18
* 7	36
* 3	72
1	144
	144
	72
	36
	18
	12
	270

JAVA

JVM, interpreted language

avoid redundancy (repetition of code)

→ control flow (jumps to method called)

↳ also means if/else/cycles/...

loops: definite (know how many times it will be executed)
indef (doesn't)

sentinel value: when seen, does smth (fencepost problem)

ARRAYS are of type reference

Types:

byte	8
short	16
int	32
long	64
float	32
double	64

short 16
bool 1

(string) array of char

△ compare string: ~~use~~ .equals()

EXCEPTION: try { // protected code
(throwable) }
catch (Exceptionname e1) {
}

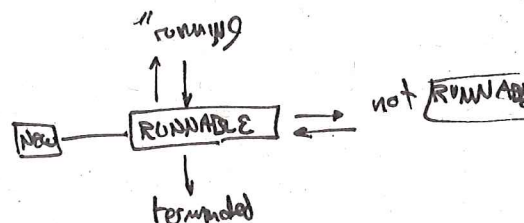
throw new Exc

es. • IllegalArgumentException
• NullPointerException
• IllegalStateException
• Arithmetic Exc

multiplexing

PCB (process control block)

implements Runnable
java.Lang. Runnable



Thread: indep. source of execution
@ override run()
call start()

Class implements runnable
you also have interrupt

priority 1-10

ExecutorService ex = ~~new~~ Executors.newFixedThreadPool(n);

ex.execute(task)

ex.shutdown()

Fork Join

ForkJoinPool fjp = new ForkJoinPool;

(thread pool, si base
su ex. service)

fjp.invoke(task)

↳ ~~(@Override)~~
compute

~~fjp~~ invokeAll(task1, task2, ...)